

Advanced Select Statement

Table Joins

To work with data that is in more than one table you need to instruct the server how to relate records from one table with its related record(s) in other tables. To do this you must use the join operator. The join operator lets us control how the server relates rows from different tables.

If you wanted to select all the staff names and their position descriptions, you would need to select the staff names from the staff table and the position descriptions from the position table.

According to the syntax of a Select statement, you could write a select statement as follows:

```
Select      Staff.FirstName, Staff.LastName, Position.PositionDescription  
From        Staff, Position;
```

When you run this select statement you return 70 records. This may seem like the correct select statement as there were no errors. However, closer inspection of the data in the two tables involved leads to some questions.

First, the Staff table contains 10 records. So, we should only see 10 records coming back (select all the staff names and their position descriptions).

Second, the position table has 7 records (there are 7 positions).

So, where did the 70 records come from? If you look at the results of the select closely you will notice that every staff member seems to have every position. $10 \text{ Staff} * 7 \text{ Positions} = 70 \text{ records}$.

What happened was there was nothing in the select statement telling the server how to relate what records in the Position table relate to any corresponding records in the Staff table.

Remember, records in tables relate to each other through the primary key and foreign key. The value of the primary key in the parent table relates to records in the child table that have the same value. Staff that are Position 2 (Program Chair) will have a value of 2 in the PositionID column which is the foreign key in the Staff Table.

Obviously, every staff member is not in every position at the school, so we need to instruct the server which column in the parent table relates to what

column in the child table. This is usually the Primary Key in one table and the Foreign Key in another. While this is not the only columns that are used in joins, they are on practically 100% of the cases. This is the primary key column in the parent table and the foreign key in the child table. In this case, it is the PositionID which joins the Position and Staff table and forms the relationship. It also identifies what records in the parent relate to what records in the child.

To ensure we get only the Parent records associated with their correct child records we must use syntax which identifies what the common (join) column is between the tables.

Syntax:

```
SELECT    column1, column2, ...
FROM      table1
          INNER JOIN table2
          ON table1.joincolumn = table2.joincolumn;
```

The Staff and Position select statement using the correct syntax:

```
Select    Staff.FirstName, Staff.LastName, Position.PositionDescription
From      Staff
          Inner Join Position
          On Staff.PositionID = Position.PositionID;
```

The Inner Join syntax tells the server that we are performing an inner join and identifies that PositionID is the column in the 2 tables that relate them together.

More specifically it performs an Inner Join which is different than an Outer Join. An Inner Join only returns records from the parent table that have related records in the child table. In this example Position is the Parent table (Primary Key in the relationship) and Staff is the child table (has the foreign key in the relationship). Looking back at the results you will notice that one Position Description is not returned in the results. Position 7 (Assistant Dean) was not returned because there are no Staff with that position. With an Inner Join, parent records that do not have a matching child record are not returned.

Outer Joins

An Outer Join is different in that it returns all the parent records, including the ones that do not have a matching record in the related child table. To

return ALL the Position Descriptions and the staff names that are in those positions you would use an Outer Join.

We will only discuss Left Outer Joins and Right Outer Joins. These both have the same purpose and you choose which one to use based on the order you list your tables in your select statement From clause.

```
Select      Staff.FirstName, Staff.LastName, Position.PositionDescription
From        Staff
           Right Outer Join Position
           On Staff.PositionID = Position.PositionID ;
```

OR

```
Select      Staff.FirstName, Staff.LastName, Position.PositionDescription
From        Position
           Left Outer Join Staff
           On Staff.PositionID = Position.PositionID ;
```

Notice the difference between the Left and Right Outer Join?

The first example states to select ALL the records from the table that is on the RIGHT side of the OUTER JOIN statement (Position) and the related records from Staff.

The second example states to select ALL the records from the table that is on the LEFT side of the OUTER JOIN statement (Position) and the related records from Staff.

Outer Joins enable us to select parent records that may not have related child records in the other table(s) in the join.

Although not necessary, joins are often coded with the parent table on the left side so you tend to see more Left Outer Join statements than Right Outer Join.

Why would you likely never see the following Select statement and why does it not make sense?

```
Select      Staff.FirstName, Staff.LastName, Position.PositionDescription
From        Staff
           Left Outer Join Position
           On Staff.PositionID = Position.PositionID ;
```

This select says select ALL the Staff records and the related records from Position. Since the Staff table is the child table, all the staff records must have a matching parent record in the Position table. Remember, all child records must have a matching parent record.

The difference between Inner Join and Outer Join:

```
Select      Student.FirstName, Student.LastName, Payment.PaymentDate
From        Student
           Inner Join Payment
           On Student.StudentID = Payment.StudentID;
```

(returns 33 records)

```
Select      Student.FirstName, Student.LastName, Payment.PaymentDate
From        Student
           Left Outer Join Payment
           On Student.StudentID = Payment.StudentID;
```

(returns 42 records)

Can you explain the difference?

The difference is there are only 33 students that have made payments (only 33 parent records that have matching child records) so only 33 records returned.

An Inner Join only returns Parent records that have matching child records.

An Outer Join returns ALL the parent records (Students) even if they do not have any matching child records (Payments). Therefore, we now know that 9 students have no payment records. Since there were no payment dates for those students the PaymentDate column for those records contains the value of NULL.

Consider the following select statement which is now selecting the StudentID column as well:

```
Select      StudentID, Student.FirstName, Student.LastName,
            Payment.PaymentDate
From        Student
            Inner Join Payment
            On Student.StudentID = Payment.StudentID;
```

This will result in an error:

ERROR: column reference "studentid" is ambiguous

This error is due to the fact that we must state the table that we are selecting the columns from. Since StudentID is in both the Student and the Payment table, the server does not know which one to select. To solve this we need to adjust our syntax to identify which table the server should select the ambiguous column from.

```
Select      Student.StudentID, Student.FirstName, Student.LastName,
            Payment.PaymentDate
From        Student
            Inner Join Payment
            On Student.StudentID = Payment.StudentID;
```

While it does not usually matter which table you select the ambiguous columns from, it is normally the parent table.

You can also encounter this error when you have an ambiguous column in other parts of a select statement (i.e. the where clause).

Selecting From 3 or More Tables

Selecting from 3 or more table is the same as selecting data that is in 2 tables. Remember the syntax for a basic Inner Join is:

```
SELECT    column1, column2,...
FROM      table1
          INNER JOIN table2
          ON table1.joincolumn = table2.joincolumn;
```

To select from more than 2 tables the syntax would look like:

```
SELECT    column1, column2, ...
FROM      table1
          INNER JOIN table2
          ON table1.joincolumn = table2.joincolumn
          INNER JOIN table3
          ON table2.joincolumn = table3.joincolumn
          INNER JOIN table4
          ON table3.joincolumn = table4.joincolumn;
```

Note: table3 and table4 may join to table1 or table2 depending on the database structure.

For every table you add to the select statement, you use a JOIN condition to add it to the select statement and indicate with the ON clause how that table relates to another table you already have in your select statement.

Select the student names, payment dates, and payment type descriptions for all the students that have payments:

```
Select    Student.FirstName, Student.LastName, Payment.PaymentDate,
          PaymentType.PaymentTypeDescription
From      Student
          Inner Join Payment
          On Student.StudentID = Payment.StudentID
          Inner Join PaymentType
          On Payment.PaymentTypeID =
          PaymentType.PaymentTypeID;
```

Table Join Summary

- Joins enable us to select data that is from different, related tables.
- We use join conditions to identify what the column is that joins each table in the relationship. This allows the select statement to return information from one table and its related information from another.
- Inner Joins only return parent records that have at least one matching child record.
- Outer Joins return all the parent records as well as any related child records.
- Left and Right Outer Joins perform the same task, and you can use either one. You must use the correct one for the order you have the tables in the select statement From clause.
- If you are selecting a column that exists in more than one table that you are selecting from you must specify which table to select from; otherwise, the database will return an ambiguous column name error. In this course, if you are writing select statements with more than one table, you **MUST** include the table name before each column, even if the column name is not duplicated.
- All the other clauses of a select statement can be used with joins.
- A join can be between 2 or more tables.
- You do not need to select a column from each table you have in your join.

Subqueries

A subquery is a select statement that is nested inside another statement. It is a full and complete select statement and can be executed on its own. We will first look at subqueries as a select statement inside another select statement. Subqueries are used in the Where or Having clause to identify the records that need to be returned.

We often refer to the subquery portion (nested inside the other select) as the Inner Query and the select statement around it as the Outer Query.

There are two types of subqueries: Nested and Correlated. In this course we focus on Nested subqueries only.

Let's look at the following select statement that uses a subquery:

```
Select      FirstName, LastName
From        Staff
Where       PositionID = (Select      PositionID
                           From        Position
                           Where       PositionDescription = 'Dean');
```

The subquery (inner query) executes once and returns one or more values and then the outer query executes using the value(s) returned from the subquery.

Execution example:

```
Select      FirstName, LastName
From        Staff
Where       PositionID = (Select      PositionID
                           From        Position
                           Where       PositionDescription = 'Dean');
```

The inner query (**subquery**) executes first and returns any values; in this case the subquery returns a value of 1 as the PositionID for 'Dean' is 1.

```
Select      FirstName, LastName
From        Staff
Where       PositionID = (1);
```


The outer query then executes using the results of the subquery as criteria for the where clause. The results of the Select executing are:

<u>FirstName</u>	<u>LastName</u>
Hugh	Guy

This is a two-step process where first the subquery returned the PositionID for the PositionDescription of 'Dean' and then the Staff Names are selected that had that PositionID. It is important to note that this example used an = in front of the subquery. If our subquery had returned more than one value it would have caused an error when the statement was executed. You can only compare one value to another one value when using an = operator.

What if the subquery returns multiple values?

One of the most common uses of a subquery is to retrieve a list of values that is used as a list of acceptable or non-acceptable criteria by the outer query.

To see all the student names that have made payments, you use the IN operator.

```
Select      FirstName, LastName
From        Student
Where       StudentID In      (Select      StudentID
                                From        Payment);
```

Breaking it down...

1. Subquery executes by first selecting all the studentID's in the payment table.
2. Then the outer query executes using those StudentID's as part of the criteria in the where clause.

What happens when you run the following Select statement?

```
Select    FirstName, LastName
From      Student
Where     StudentID =      (Select    StudentID
                           From      Payment);
```

ERROR: more than one row returned by a subquery used as an expression

Remember, when using = there must be only one value on each side. If the subquery might return more than one value, then you must use the IN operator.

What if I wanted to see all students that have never made a payment?

```
Select    FirstName, LastName
From      Student
Where     StudentID Not In (Select    StudentID
                           From      Payment);
```

Additional Operators

In the previous example when the subquery selects all the StudentID's from the payment table you end up with a lot of duplicate StudentID values. This may seem inefficient when you could write the subquery like this:

```
Select      FirstName, LastName
From        Student
Where       StudentID In      (Select      Distinct StudentID
                               From        Payment);
```

Remember, distinct only returns unique values and not duplicate values. Testing shows that using distinct in the subquery to return unique values is actually slower. Keep in mind that there is some overhead in the selecting of only distinct values versus simply selecting all of the values.

Any/Some and All

So far, our subqueries have returned values to be used as criteria for an exact match by the outer query.

Where StudentID IN (Select StudentID)

What if we want something other than an exact match?

The ANY or SOME (for SQL they are interchangeable) or ALL operators lets you use even more operators (>, <, >=, <=, <>).

ANY/SOME compares against any of the values retrieved by the subquery.

Where Grade > ANY (Select Grade.....)

If the Grade being selected by the outer query is greater than **any** of the grades returned by the subquery, then the record is returned.

The ALL operator compares against all of the values retrieved by the subquery.

Where Grade > ALL (Select Grade.....)

If the Grade being selected by the outer query is greater than **all** of the grades returned by the subquery, then the record is returned.

We can use these operators to perform some more complicated select statements. Subqueries can be used in many places where you need to retrieve a set of values for criteria.

Select the City that has the highest number of students in it.

```
Select      City
From        Student
Group By City
Having      Count (StudentID) >= All      (Select      Count (StudentID)
                                           From        Student
                                           Group By City);
```

In this example we used the subquery as criteria in the having clause.

Breaking it down:

1. The subquery executes first and returns values (1,1,2,1,8,1,1,1) and the outer query is re-written with those values.

```
Select      City
From        Student
Group By City
Having      Count (StudentID) >= All (1,1,2,1,8,1,1,1);
```

2. The outer query then executes comparing the count of students in each city to see which count(s) are greater/equal than or equal to all the ones in the subquery. The record from the outer query that is equal to the highest value in the subquery is the city with the most students.

Unions

The union operation lets you combine the data retrieved by multiple select statements into a single result table.

Syntax:

```
SELECT . . .
```

```
UNION [ALL]
```

```
SELECT . . .
```

Let's write a query to return all the student names and combine it (UNION) with all the staff names.

```
Select      FirstName, LastName
From        Student
Union
Select      FirstName, LastName
From        Staff;
```

This results in 25 records being returned. There are 16 students and 10 staff. Why are we missing one? UNION omits duplicates unless you use UNION ALL. You can UNION as many queries together as you need/want to.

There are some rules to follow, however:

1. All the queries being combined with UNION must return the same number of columns.

```
Select      FirstName, LastName, City
From        Student
Union
Select      FirstName, LastName
From        Staff;
```

ERROR: each UNION query must have the same number of columns

2. The columns being combined must share similar datatypes.

```
Select      FirstName, Birthdate
From        Student
Union
Select      FirstName, LastName
From        Staff;
```

ERROR: UNION types date and character varying cannot be matched

3. Ordering the results must be done after the last query.

```
Select      FirstName, LastName
From        Student
Union
Select      FirstName, LastName
From        Staff
Order By LastName ASC;
```

Works!

4. Column names are defined in the first query.

```
Select      FirstName "First Name", LastName "Last Name"
From        Student
Union
Select      FirstName, LastName
From        Staff
Order By "Last Name" ASC;
```

Works!

Views

A view is a query that is stored in the database. It behaves very much like a virtual table that was created by selecting rows and columns from other tables. In most situations it can be used just like an actual table. There is no real overhead with creating a view, as there is no actual data stored in the view. It is created by select statement(s) and the data remains in the original tables.

Some common uses for SQL views are:

- simplifying data retrieval from complex queries
- hiding the underlying table structure
- controlling access to data for different users

Simplifying Data Retrieval

As you may be starting to notice, sometimes we need to code complicated select statements containing joins, aggregates, function, unions, etc. to produce the results we want. A view can simplify this task. Some or all of a complex select statement can be created as a view and future queries of that information can be executed against the view.

For example, if we were often executing queries involving publishers and their titles, we would constantly be writing selects with a join to retrieve data from those tables and then executing additional criteria in the where clause to retrieve more specific data. A view could be created which returns all the publishers and their titles. Any specific queries about that data could then be executed against the view. This would allow us to not have to code a join condition every time we wanted information from those tables together as we have already created a view that contains that information.

Hiding The Underlying Table Structure

By creating a view, we create an additional layer between the user and the tables. If the definition of a table changes only the view needs to be altered to accommodate the changes. As long as the view contains the same data as it did before the changes to the underlying table(s), then the user is unaware of the changes, and they are of no concern to them.

A common example of this is when a table gets too big and some of the records are moved to an archive table. If a view was being used that contained all the rows and records of the original table it could now be altered to select all the records from the original table and UNION them with

the records from the archive table. As far as the user of the view is concerned, the view contains the same records as before. The definition of the view had to be changed to select the records from both the original table and the archive table, but the result is the same. The view contains all the records that were there originally.

Controlling Access to Data

By creating a view, you can restrict which columns certain users are able to see. Suppose that some information in an employee table should be available to everyone, but some columns are restricted information. A view can be created that contains the columns that are available to everyone, and permissions can be granted on that view to everyone. Other views could contain other combinations of columns and permissions that could be granted accordingly to those views for the appropriate users and groups.

Data Modification

A user can do more than just select data from views. As mentioned before, a view can be treated like a table (with some exceptions). You can update, insert, and delete records in a view just as if you were doing so on an actual table. Any changes to the view are reflected in the records in the table that make up the view.

Here are some points to keep in mind when modifying records through a view in PostgreSQL:

- Modifying records through a view must meet all rules and constraints that were created on the tables that the view is based on.
- You cannot insert, update, or delete from a view that is based on more than one table.
- If the view select statement contains distinct, group by, having, aggregate functions, or includes a union, you cannot modify data with that view.

Create a View

SYNTAX:

```
CREATE [OR REPLACE] VIEW viewname
```

```
AS
```

```
Select ...
```

Note: a view cannot reference more than 1600 columns (depending on column types)

Create View StudentCourseView

As

```
Select      Student.FirstName, Student.LastName, Course.CourseID,  
            Course.CourseName  
From        Student  
            Inner Join Registration  
                On Student.StudentID = Registration.StudentID  
            Inner Join Course  
                On Course.CourseID = Registration.CourseID;
```

What can you do with this view? Try selecting data from it.

```
Select      FirstName, LastName, CourseName  
From        StudentCourseView;
```

Works.

Insert into StudentCourseView

(FirstName, LastName, CourseID, CourseName)

Values

('Bob', 'Smith', 'SDEV1201', 'New Database Course');

Fails. You cannot insert data through a view if the view is based on multiple tables.

ERROR: cannot insert into view "studentcourseview"

Views that do not select from a single table or view are not automatically updatable.

Update StudentCourseView

Set CourseName = 'ChangedCourseName'

Where CourseID = 'DMIT1508';

Fails. You cannot Update through a view if the view is based on more than one table.

```
ERROR: cannot update view "studentcourseview"  
Views that do not select from a single table or view are not automatically  
updatable.
```

```
Delete From StudentCourseView  
Where CourseID = 'DMIT1508';
```

Fails. You cannot Delete through a view if the view is based on more than one table.

```
ERROR: cannot delete from view "studentcourseview"  
Views that do not select from a single table or view are not automatically  
updatable.
```

Create a one table view.

```
Create View GenericRegistrationInformation  
As  
Select      StudentID, CourseID, Semester  
From        Registration;
```

Try to insert a student using the StudentCourseView.

```
Insert into GenericRegistrationInformation  
      (StudentID, CourseID, Semester)  
Values  
      (200578400, 'DMIT1508', '2005J');
```

Succeeds. The record is inserted successfully because the view is based on a single table.

Now attempt to delete that student.

```
Delete From GenericRegistrationInformation  
Where      StudentID = 200578400 and  
           CourseID = 'DMIT1508' and  
           Semester = '2005J';
```

Succeeds. The record is deleted successfully because the view is based on a single table.

Dropping Views

Views can be dropped using the DROP keyword.

SYNTAX

```
DROP VIEW [IF EXISTS] viewname;
```

Correlated Subqueries

Note: Correlated Subqueries are not assessed in SDEV1201.

Correlated subqueries are a bit different than the subqueries we have looked at previously. Those were called nested subqueries. Nested subqueries had the subquery query execute once and then the outer query executed once.

Correlated subqueries work like this:

1. The outer query retrieves a record.
2. The inner query executes using data passed to it from the record retrieved by the outer query.
3. The inner query passes data back to the outer query. This data is used by the outer query to finish processing the record.

The process repeats once for every record processed by the outer query.

The defining feature of a correlated subquery is that the inner query references (i.e. uses data from) the outer query. Simply put, the inner query “looks” at the outer query.

The following example uses a correlated subquery to find the students whose BalanceOwing is greater than the average payment that student has made.

```
Select      Student.StudentID, Student.FirstName, Student.LastName,
            Student.BalanceOwing
From        Student
Where       Student.BalanceOwing > (
            Select      AVG(Payment.Amount)
            From        Payment
            Where       Payment.StudentID = Student.StudentID);
```

Example 2: Find students whose last payment is less than \$500.00

```
Select      Student.StudentID, Student.FirstName, Student.LastName,
            Student.BalanceOwing
From        Student
Where       Student.StudentID In (
            Select      Payment.StudentID
            From        Payment
            Where       Payment.Amount < 500 and
                       Payment.StudentID = Student.StudentID);
```

Cross Joins

Note: Cross Joins are not assessed in SDEV1201.

The Cross Join produces a Cartesian product of two tables, meaning it returns all possible combinations of rows from the joined tables. This basic syntax is used when you need to combine every row of one table with every row of another table.

Cross Joins are used infrequently but are especially useful for generating datasets for analysis. An administrator could match every student with every club to generate what full club membership would look like.

```
Select      Student.StudentID, Student.FirstName, Student.LastName,
            Club.ClubName
From        Student
            Cross Join Club;
```

In this scenario, there are 16 students and 8 clubs resulting in 128 records returned ($16 * 8 = 128$).

-- Create a select statement to display every student with every type of payment.

```
Select      Student.StudentID, Student.FirstName, Student.LastName,
            PaymentType.PaymentTypeDescription
From        Student
            Cross Join PaymentType pt
Order By Student.StudentID, PaymentType.PaymentTypeDescription;
```

In this scenario, there are 16 students and 6 payment types resulting in 96 records returned ($16 * 6 = 96$).